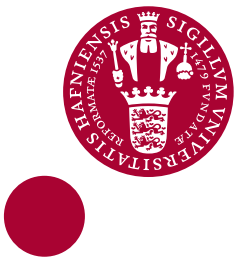


Passive and partially active fault tolerance for massively parallel stream processing engines

Su, Li; Zhou, Yongluan

Published in:
IEEE Transactions on Knowledge and Data Engineering

DOI:
10.1109/TKDE.2017.2720602

Publication date:
2019

Passive and Partially Active Fault Tolerance for Massively Parallel Stream Processing Engines

Li Su, Yongluan Zhou

Abstract—Fault-tolerance techniques for stream processing engines can be categorized into passive and active approaches. However, both approaches have their own inadequacies in Massively Parallel Stream Processing Engines (MPSPE). The passive approach incurs a long recovery latency especially when a number of correlated nodes fail simultaneously, while the active approach requires extra replication resources. In this paper, we propose a new fault-tolerance framework, which is Passive and Partially Active (PPA). In a PPA scheme, the passive approach is applied to all tasks while only a selected set of tasks will be actively replicated. The number of actively replicated tasks depends on the available resources. If tasks without active replicas fail, tentative outputs will be generated before the completion of the recovery process. We also propose effective and efficient algorithms to optimize a partially active replication plan to maximize the quality of tentative outputs. We implemented PPA on top of Storm, an open-source MPSPE and conducted extensive experiments using both real and synthetic datasets to verify the effectiveness of our approach.

Index Terms—Distributed Stream Processing, Fault Tolerance.

1 Introduction

There is a recently emerging interest in building Massively Parallel Stream Processing Engines (MPSPE), such as Storm [26], and Spark Streaming[28], which make use of large-scale computing clusters to process continuous queries over fast data streams. Such continuous queries often run for a very long time and would unavoidably experience various system failures, especially in a large-scale cluster. As it is critical to provide continuous query results without significant downtime in many data stream applications, fault-tolerance techniques in Stream Processing Engines (SPEs) [3], [5], [28] have attracted a lot of attention.

Existing fault-tolerance techniques for SPEs can be generally categorized as passive and active approaches [13]. In a typical passive approach, the runtime states of tasks will be periodically extracted as checkpoints and stored at different locations. Upon failure, the state of a failed task can be restored from its latest checkpoint. While one can in general tune the checkpoint frequency to achieve trade-offs between the cost of checkpoint and the recovery latency, the checkpoint frequency should be limited to avoid high checkpoint overhead, which affects the system performance. Hence recovery latency is usually significant in a passive approach. When one wants to minimize the recovery latency as much as possible, it is often more efficient to use an active approach, which typically uses one backup node to replicate the tasks running on each processing node. When a node fails, its backup can quickly take over with minimal latency.

Even though there are abundant fault-tolerance techniques in SPEs, developing an MPSPE [26] poses great challenges to the problem. First of all, in a large cluster, there are often two different types of failures: independent failure and correlated failure [10], [20]. Previous studies mostly focused on independent failure that happens at a single node. Correlated failures are usually caused by failures of switches, routers and power facilities, and will involve a number of nodes failing simultaneously. With such failures, one has to recover a large number of failed tasks and temporarily run them on an additional set of standby nodes before the failed ones are recovered. Using a passive fault tolerance approach, one has to keep the standby nodes running even their utilization is low most of the time in order to avoid the unacceptable overhead of starting them at recovery time. Furthermore, as checkpoints of different nodes are often created asynchronously, massive synchronizations have to be performed during recovery. Therefore it could be difficult to meet the user requirements on recovery latency even with a relatively high checkpoint frequency.

On the other hand, while an active fault-tolerance approach can achieve a lower recovery latency, it could be too costly for a large-scale computation. Consider a large-scale stream computation that is parallelized onto 100 nodes, one may not be able to afford another 100 backup nodes for active replication.

Another challenge is that there exist some time-critical applications which prefer query outputs being generated in good time even if the outputs are computed based on incomplete inputs. This kind of applications usually require continuous query output for real-time opportune decision-making or visualization. Consider a community-based navigation service, which collects and aggregates user-contributed traffic data in a real-time fashion and then continuously provides navigation suggestions to the users. Failure of some processing nodes could result in losing some user-contributed data. The system, while waiting for the failed nodes to recover, can continue to help drivers plan their routes based on the incomplete inputs. Other examples of such applications are like intrusion detections,

- *Li Su is with the Department of Mathematics and Computer Science, University of Southern Denmark, E-mail: lsu@imada.sdu.dk*
- *Yongluan Zhou is with the Department of Computer Science, University of Copenhagen, E-mail: zhou@di.ku.dk*

online visualization of real-time data streams etc. Alerts of events matching the intrusion attack patterns or infographics generated over incomplete inputs are still meaningful to the users and should be generated without any major delay. Consider the long recovery latency for a large-scale correlated failure, the lack of trade-offs between recovery latency and result quality would not be able to fulfill the requirements of these applications.

To address the aforementioned challenges, we propose a new fault-tolerance scheme for MPSPEs, which is Passive and Partially Active (PPA). In a PPA scheme, a number of standby nodes will be used to prepare for recoveries from both independent and correlated failures. Checkpoints of the processing nodes will be stored at the standby nodes periodically. Rather than keeping them mostly idled as in a purely passive approach, we opportunistically employ them for active replications for a selected subset of the running tasks. In this way, we can provide very fast recovery for the tasks with active replicas. Furthermore, when the failed tasks contain those without active replicas, PPA provides *tentative* outputs with quality as high as possible. The results can then be rectified after the passive recovery process has been finished using similar techniques proposed in [3]. In general, PPA is more flexible in utilizing the available resources than a purely active approach, and in the meantime can provide tentative outputs with a higher quality than a purely passive one.

In this paper, we focus on optimizing utilizing available resources for active replication in PPA, i.e. deciding which tasks should be included for active replication. In summary, we have made the following contributions in this paper:

(1) We present PPA, a passive but partially active fault-tolerance scheme for a MPSPE.

(2) As existing MPSPEs often involve user defined functions whose semantics are not easily available to the system, we propose a simple yet effective metric, referred to as output fidelity, to estimate the quality of the tentative outputs.

(3) We propose a structure-aware algorithm to determine which tasks to actively replicate for single-query topology. Furthermore, we extend our solution to support the optimization for multi-query topology.

(4) For the passively replicated tasks in a PPA plan, we propose incremental recovery to schedule their recoveries to optimize the quality of tentative outputs.

(5) We implement our approach in an open-source MPSPE, namely Storm [26] and perform an extensive experimental study on an Amazon EC2 cluster using both real and synthetic datasets. The results suggest that by adopting PPA, the accuracy of tentative outputs are significantly improved with limited amount of replication resources.

This paper is an extended version of [25]. The extension includes new techniques to incorporate multi-query topologies and incremental recovery.

2 System Model

2.1 Data and Query Model

As in existing MPSPEs [26], we assume that a data item is modeled as a key-value pair. Without loss of generality, the key of a data item is assumed to be a string and the value is a blob in an arbitrary form that is opaque to the system.

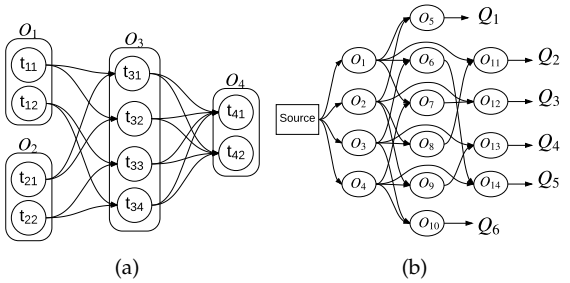


Fig. 1. (a): A query plan that consists of 4 operators (O_1, O_2, O_3, O_4) with different numbers of tasks. (b): An example multi-query topology that consists of 6 queries with operator sharing.

A query execution plan in MPSPEs typically consists of multiple operators, each being parallelized onto multiple processing nodes based on the key of input data. Each operator is assumed to be a user-defined function. We model such query plan as a topology of the parallel tasks of all the operators. By modeling each task as a vertex and the data flow between each pair of tasks as a directed edge, the topology can be represented as a Directed Acyclic Graph (DAG). Figure 1(a) shows an example of topology. Each task represents the workload of an operator that is assigned to a processing node in the cluster and all the tasks that belong to the same operator will conduct the same computation.

An operator can subscribe to the outputs from multiple operators except for itself. The output stream of every task is partitioned into a set of substreams using a particular partitioning function, which divides the keys of a stream into multiple key partitions and splits the stream into substreams based on these key partitions. For each task, the input substreams received from the tasks within an upstream neighboring operator will constitute an input stream. Therefore, the number of input streams of a task is up to the number of its upstream neighboring operators.

Similar to [30], we consider the following four common partitioning situations between two neighboring operators in a MPSPE. In the following descriptions, we consider an upstream operator containing N_1 tasks and a downstream operator containing N_2 tasks.

- *One-to-one*: each upstream task only sends data to a single downstream task and a downstream task only receives data from a single upstream task.
- *Split*: each upstream task sends data to M_2, $2 \leq M_2 < N_2$, downstream tasks and each downstream task only receives data from a single upstream task.
- *Merge*: each upstream task sends data to only one downstream task and each downstream task receives data from M_1, $2 \leq M_1 < N_1$, upstream tasks.
- *Full*: each upstream task sends data to all N_2 downstream tasks.

2.2 Multi-Query Topology Model

It is common that the stream application running on MPSPEs consists of multiple queries, which could share the computation results of operators with each other. Therefore, we extend the single query topology model in [25] to support multiple queries being executed concurrently. Figure 1(b) depicts an example of multi-query topology which consists of 6 queries, where the outputs of operators can be used by multiple queries. For query Q_i, a user-specified priority prt_i is assigned, which is a nonnegative

integer.

2.3 PPA Replication Plan

Given a topology T and its whole set of tasks $\mathcal{M}$, a PPA replication plan for T consists of two parts: a passive replication plan that covers all the tasks in $\mathcal{M}$ and a partially active replication plan which covers a subset of $\mathcal{M}$, denoted as $\mathbf{P}$. With the passive replication plan, checkpoints will be periodically created for all the tasks and stored at the standby nodes. For a task t_i, its checkpoint consists of t_i's computation state and output buffer. After a checkpoint is extracted from t_i, its upstream neighboring tasks will be notified to prune the unnecessary data from their output buffers. The buffer trimming should guarantee that, if t_i fails, its computation state can be recovered by loading its latest checkpoint and replaying the output buffers in its upstream tasks. On the other hand, for each $t_i \in \mathbf{P}$, an active replica will be created, which will receive the same input and perform the same processing as t_i's primary copy.

Upon failures, the actively replicated tasks will be recovered immediately using their active replicas, meanwhile the tasks that are only passively replicated will be restored from their latest checkpoints. When there are some failed tasks belonging to $\mathcal{M}-\mathbf{P}$, tentative outputs will be produced before they are fully recovered. Such tentative outputs have a degraded quality due to the loss of input data that otherwise should be processed by the failed tasks belonging to $\mathcal{M}-\mathbf{P}$.

In the next section, firstly we present how to estimate the quality of tentative outputs of queries. Then we propose the formal problem statement on optimizing the partially active replication plan to maximize the quality of tentative outputs in multi-query topology.

3 Problem Formulation

3.1 Quality of Tentative Outputs

Previous works on load shedding [2], [16] have studied how to evaluate the quality of query outputs in case of lost of input data. Their models assume full knowledge of the semantics of individual operators and hence can estimate the output quality in a relatively precise way. However, in existing MPSPEs, such as Storm, operators are often opaque to the system and may contain complex user-defined functions written in imperative programming languages. The existing models therefore cannot be easily applied. In our first attempt, we have tried to derive output accuracy models composed by some generic functions, which should be chosen or provided by the users according to the semantics of the operators. We found that this approach is not very user friendly and it may be very difficult for a user to provide such functions for a complicated operator.

Therefore, we strive to design a model that requires users to provide minimum information of an operator's semantic, but yet is effective in estimating the quality of tentative outputs. More specifically, we propose a metric, called *Output Fidelity* (OF), which is roughly equal to the ratio of the source input that can contribute to tentative outputs of each query. This is based on the assumption that the accuracy of query's tentative outputs increases with more complete input and a PPA plan with a higher OF value would incur more accurate tentative outputs.

In the rest of this section, we first present the details of estimating OF for each single query in the topology, after that we present the formal problem formulation on optimizing the PPA plan for the entire multi-query topology.

3.1.1 Operator Output Loss Model

Upon task failures, we need to propagate the information losses incurred by any failed task to the output of the sink operator. Suppose task t_{22} in Figure 2 is failed, we need to transform the input loss of t_{31} into its output loss. In this subsection, we propose the operator output loss model, which estimates the information loss of an operator's output based on the information loss of its input. In the next subsection, we present the precise definition of OF.

In following descriptions, the set of input streams of task t_i are denoted as $\left\{S_{i,1}^{in}, S_{i,2}^{in}, ..., S_{i,p}^{in}\right\}$, where the rate of $S_{i,j}^{in}$ is represented as $\lambda_{i,j}^{in}$ and its information loss is referred to as $IL_{i,j}^{in}$. The rate of t_i's output stream, S_i^{out}, is referred to as λ_i^{out}, and its information loss is denoted as IL_i^{out}. If t_i is failed, its output will be lost and IL_i^{out} will be set as 1. Otherwise, we calculate IL_i^{out} based on the information losses of t_i's input streams.

As described in the query model, an input stream of a task may consist of multiple substreams, which are sourced from tasks belonging to the same upstream neighboring operator. Suppose that $S_{i,j}^{in}$ consists of a set of substreams $U_{i,j}^{in}$. For each substream s_k, $s_k \in U_{i,j}^{in}$, denoting its rate as λ_{s_k} and its information loss as IL_{s_k}, then the information loss of $S_{i,j}^{in}$ is calculated as:

$$IL_{i,j}^{in} = \frac{\sum_{s_k}^{s_k \in U_{i,j}^{in}} \lambda_{s_k} \cdot IL_{s_k}}{\sum_{s_k}^{s_k \in U_{i,j}^{in}} \lambda_{s_k}} \tag{1}$$

Meanwhile, the output stream of task t_i, S_i^{out}, can be split into a set of substreams, denoted as D_i^{out}. For each substream s_k belonging to D_i^{out}, its information loss is estimated to be equal to S_i^{out}, i.e. $IL_{s_k} = IL_i^{out}$.

Figure 2 depicts an example query plan as well as the rate of each output stream. IL_{31}^{out} represents the information loss of output stream S_{31}^{out} caused by the failure of task t_{22}. We distinguish two situations and use this example to illustrate the calculation of information loss of a task's output stream.

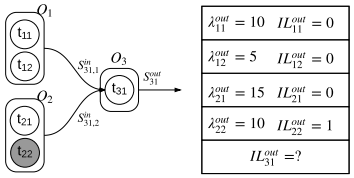


Fig. 2. An illustrating query plan with task failure, where $\lambda_{31,1}^{in} = \lambda_{11}^{out} + \lambda_{12}^{out}$ and $\lambda_{31,2}^{in} = \lambda_{21}^{out} + \lambda_{22}^{out}$.

Correlated-Input Operator. O_t performs computations over the join results of its input streams. For example, suppose O_3 in Figure 2 is a join operator. Without further semantic information of O_3, we consider the effective input of t_{31} as the Cartesian product of its input streams, whose rate is equal to $\left(\lambda_{31,1}^{in} \cdot \lambda_{31,2}^{in}\right)$ and its information loss can be computed as $\left[1 - \left(1 - IL_{31,1}^{in}\right) \cdot \left(1 - IL_{31,2}^{in}\right)\right]$. By assuming that the information loss of t_{31}'s output should be equal to that of its input stream, we can get $IL_{31}^{out} = \frac{2}{5}$.

In summary, the information loss of t_i's output stream can be calculated as:

$$IL_i^{out} = 1 - \prod_{j=1}^{p} \left(1 - IL_{i,j}^{in}\right) \quad (2)$$

Independent-Input Operator. O_t does not compute joins over input streams. If O_3 in Figure 2 is an independent-input operator, the effective input of t_{31} is considered as the union of its input streams, whose rate is equal to $\left(\lambda_{31,1}^{in} + \lambda_{31,2}^{in}\right)$ and its input loss can be calculated as $\frac{\lambda_{31,1}^{in} \cdot IL_{31,1}^{in} + \lambda_{31,2}^{in} \cdot IL_{31,2}^{in}}{\lambda_{31,1}^{in} + \lambda_{31,2}^{in}}$. Similar to the correlated-input operator, we also assume that the information loss of t_{31}'s output should be equal to that of its input stream. Then we have, in this example, $IL_{31}^{out} = \frac{1}{4}$. In general, the information loss of t_i's output stream can be calculated as follows:

$$IL_i^{out} = \frac{\sum_{j=1}^{p} \lambda_{i,j}^{in} \cdot IL_{i,j}^{in}}{\sum_{k=1}^{p} \lambda_{i,k}^{in}} \quad (3)$$

Recall that one of the design principles is to request as little information of the operators' semantics as possible. We distinguish the aforementioned two types of operators simply because the characteristics of their effective inputs are very different. With such distinction, the OF metric can be estimated much more precisely.

3.1.2 Output Fidelity

With the operator output loss model, the output information losses of tasks in the sink operator can be calculated by conducting a depth-first traversal of the query plan, which starts from the tasks in the source operators and ends at the tasks in the sink operator.

By denoting the sink operator of query plan Q_i in topology T as O_{sink}, and the set of tasks belonging to O_{sink} as $\{t_1, t_2, ..., t_{M_i}\}$, The output fidelity of query plan Q_i, OF_i, is defined as:

$$OF_i = 1 - \frac{\sum_{j=1}^{M_i} \lambda_j^{out} \cdot IL_j^{out}}{\sum_{k=1}^{M_i} \lambda_k^{out}} \quad (4)$$

With equation 4, the output fidelity of the query in Figure 2 could be calculated as $OF = 1 - \frac{\lambda_{31}^{out} \cdot IL_{31}^{out}}{\lambda_{31}^{out}}$. Suppose that O_3 in this example is an independent-input operator and $IL_{31}^{out} = \frac{1}{4}$, we have $OF = \frac{3}{4}$.

3.2 Problem Statement

Before presenting the problem definition, we introduce a concept: *Minimal Complete Tree*, which is also referred to as *MC-Tree* for simplicity in the following sections.

Definition 1. MINIMAL COMPLETE TREE (MC-TREE)*: A minimal complete tree is a tree-structured subgraph of the query plan. The source vertices of this subgraph correspond to tasks from the source operators of the query and its sink vertex is a task from an output operator of this query.*

A minimal complete tree can continuously contribute to final query outputs if and only if all its tasks are alive. Taking the query plan in Figure 1(a) for instance, O_4 must be an independent-input operator as it has only one upstream neighboring operator. Each task in O_4 has 4 input streams, these is no correlation among these input streams as O_4 is an independent-input operator. If O_3 is an independent-input operator, tasks in $\{t_{11}, t_{31}, t_{41}\}$ can constitute an MC-Tree and there are 16 MC-Trees in the query plan. However, if O_3 is a correlated-input operator, t_{31} cannot produce any output if either t_{11} or t_{21} fails. Hence tasks in $\{t_{11}, t_{21}, t_{31}, t_{41}\}$ can constitute an MC-Tree and there are in total 8 MC-Trees in the query plan.

Based on Definition 1, if failures of tasks in an MC-Tree occur, it will only continue propagating data to the sink operator if and only if all of its failed tasks are actively replicated. Suppose topology T consists of a set of queries $Q_1, Q_2, ..., Q_N$ and the available resources can be used to actively replicate R tasks ($R \leq |\mathcal{M}|$, where $\mathcal{M}$ is all the tasks of T), then the problem of optimizing a partially active replication plan is defined as follows:

Definition 2. OPTIMIZING PPA PLAN*: Given a topology T, denoting the priority of query Q_i as prt_i, the output fidelity of query Q_i in the partial topology composed by the actively replicated MC-Trees in T is denoted as OF_i. Actively replicate a set of tasks under the resource constraint R such that the value of $\sum_{i=1}^{N} prt_i \cdot OF_i$ is maximized.*

This problem is NP-hard, as it can be reduced from the Set-Union Knapsack Problem [8] in polynomial time, which is NP-hard.

4 ACTIVE REPLICATION OPTIMIZATION

The dynamic programming (DP) algorithm presented in [25] searches for the optimal PPA plan by selecting a subset of MC-Trees for replication under the resource constraint to maximize OF. Due to the high time complexity of DP, we introduce two heuristic algorithms designed for single-query topology. Then we present how to handle multi-query topology as an extension to the work in [25].

Inspired by the DP algorithm [25], we design a structure-aware algorithm that, at each step, rather than enumerates all the possible expansions of a candidate plan, only expands it with an MC-Tree that can incur the greatest increase in OF per resource unit. Unfortunately, even such a greedy approach may fall short under the following situation. Consider a topology T that consists of a sequence of k operators and all the operators use Full partitioning, the number of MC-Trees within T is equal to $\prod_{i=1}^{k} M_i$, where M_i is the number of tasks of operator O_i. In such a topology, the number of MC-Trees will grow very fast with increasing number of operators. Therefore, even a greedy search among the possible combinations of MC-Trees would not perform well.

To solve this problem, we decompose a general topology into two types of topologies, namely full topologies and structured topologies whose definitions are as follows, and then optimize them separately.

- **Structured topology** is defined as a topology where only the operators, that produce outputs of this topology, can have a Full partitioning function and the others have other types of partitioning functions.
- **Full topology** is defined as a topology that all of its operators have a Full partitioning function.

The rest of this subsection is organized as follows: firstly, we present the algorithms generating PPA plans for structured topologies and full topologies, respectively. Then we

Algorithm 1: PLANSTRUCTUREDTOPOLOGY($\mathbf{P}, R, T$)

Input: An initial plan $\mathbf{P}$; The amount of available resources R; Topology T;
Output: Replication plan $\mathbf{P}$;

```
1  usage = 0; S_u ← Set of the units split from topology T;
2  foreach Unit U_i ∈ S_u do
3  |  Build segment set G_i;
4  while usage ≤ R do
5  |  Candidates ← ∅ ;
6  |  foreach Unit U_i ∈ S_u do
7  |  |  foreach non-replicated segment g_i ∈ U_i do
8  |  |  |  CG_i ← {g_i};
9  |  |  |  if OF_P = OF_{P∪CG_i} then
10 |  |  |  |  Conduct a BFS from U_i to traverse all the units:
11 |  |  |  |  foreach visited unit U_j during the BFS do
12 |  |  |  |  |  Segment g_j ← max_of (U_j) ;
   |  |  |  |  |  /* max_of (U_j) returns the segment in
   |  |  |  |  |     U_j, which is connected with
   |  |  |  |  |     segment in CG_i and has the
   |  |  |  |  |     maximal OF with U_j treated as an
   |  |  |  |  |     independent topology;          */
13 |  |  |  |  |  if Cost(CG_i) + Cost(g_j) ≤ usage then
14 |  |  |  |  |  |  CG_i = CG_i ∪ g_j;
15 |  |  |  |  |  else Stop the BFS ;
16 |  |  |  Candidates ← Candidates ∪ CG_i;
17 |  Find CG_opt from Candidates such that the following value is
   |    maximized: (OF_{P∪CG_opt} − OF_P)/Cost(CG_opt);
18 |  P ← P ∪ CG_opt; usage ← usage + Cost(CG_opt);
19 |  if CG_opt ≠ ∅ then return P;
20 |  Remove the completely replicated units from S_u;
21 Return P;
```

explain the structure-aware algorithm, which generates PPA plan for a general topology by decomposing it into several sub-topologies, each being either a structured topology or a full topology.

4.1 Algorithm for Structured Topology

Although we define structured topology such that Full partitioning only exists in the output operators, the number of MC-Trees in a structured topology could still be very large. Consider the situation that a task t_i receives N_{in} input streams and produce N_{out} output streams, there will be at least $N_{in} * N_{out}$ MC-Trees containing t_i. In addition, if t_i joins N_k substreams from operator O_k with N_j substreams from operator O_j, the number of MC-Trees containing t_i will at least be equal to $N_k \cdot N_j$. To avoid bad performance due to the large number of MC-Trees, we split a structured topology into multiple units such that, within a unit, the number of MC-Trees is equal to the maximal number of input substreams among the operators within this unit. We refer to an MC-Tree in a unit as *segment* to differentiate it from a complete MC-Tree in the topology.

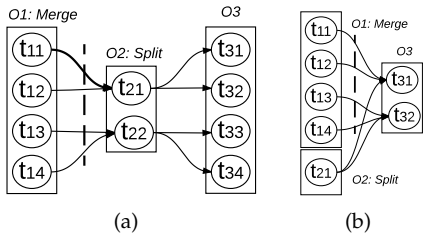


Fig. 3. Split structured topologies into units. O_3 in Figure 3(b) is a join operator. The dashed line denotes the unit boundary.

The situation of multiple input streams and multiple output streams occurs on the task who has an input stream partitioned with Merge and an output stream partitioned with Split, a unit boundary will be set between this operator and its upstream neighboring operator using Merge partitioning. For instance, a unit boundary is set between O_1 and O_2 in the topology in Figure 3(a), such that there are 4 MC-Trees in both unit $\{O_1\}$ and unit $\{O_2, O_3\}$. The situation that a task joins multiple input substreams from one operator with substreams from other operators happens on the tasks of join operators that have at least one input stream partitioned with Merge. As illustrated in Figure 3(b), a unit boundary is set between O_1 and O_3 and the splitting generates 2 units, which are unit $\{O_1\}$ that has 4 MC-trees and and unit $\{O_2, O_3\}$ that consists of 2 MC-Trees. Note that, within a structured topology, replicating a segment is beneficial only if all the other segments within the same complete MC-Tree are also replicated. In other words, we should avoid enumerating plans that replicate a set of disconnected segments.

The details of the algorithm for structured topology are presented in Algorithm 1. The algorithm searches through the units generated from input topology. Within unit U_i, if the set of non-replicated segments is not empty, we check whether replicating these segments will increase the final output accuracy (line 9). Note that this will only be true if this segment can form a complete MC-Tree with the other replicated segments within the current plan. Each of such segments will be put into a candidate pool (line 16). If the segment g_i does not enhance the plan's OF, we conduct a BFS (Breadth-first search) starting from U_i and traversing through all the units in Topology T. The BFS is terminated until $usage$ is less then the non-replicated tasks in CG_i. Finally, every unit visited during the BFS contributes a segment to CG_i and the segments from neighboring units are connected (lines $10-15$). Then we put such a set of segments as one candidate in the candidate pool.

After finishing the scanning of all units, we get a candidate pool consisting of a number of segment sets, each containing one or more segments. We use a profit density function to rank the candidates. The profit density of a candidate CG_k is calculated as $(OF_{P\cup CG_k} - OF_P)/Cost(CG_k)$, where OF_P is the OF value of plan P, $OF_{P\cup CG_k}$ is the OF value after expanding P by replicating segment in CG_k. $Cost(CG_k)$ is the resource consumption of non-replicated tasks within CG_k. The plan in the candidate pool with the maximum profit density will be merged with the input plan P and returned. The complexity of Algorithm 1 is equal to $O(R \cdot N \cdot M^2 \cdot E)$, where R is the amount of available replication resources, N is the number of operators, M represents the average degree of parallelization of operators in T, and E is the number of neighboring unit pairs.

4.2 Algorithm for Full Topology

Each task within a full topology will send input data to all the tasks that belong to its downstream neighboring operators. We propose an algorithm for full topology as illustrated in Algorithm 2. The basic idea of this algorithm is that, within any operator, we always prefer to replicate the task that will bring the maximum increase of OF under the assumption that all the other tasks that belong to the same operator are failed and the tasks that belong to other operators are alive. We denote the increase of OF by replicating task t_{ij} as δ_{ij}. If the input plan P is empty, we first select one task from each operator that has the largest δ_{ij} among all the tasks in this operator and put it into P (lines $5-6$). If P is not empty, we iterate and select R tasks that have

Algorithm 2: PLANFULLTOPOLOGY($\mathbf{P}$, R, T)

Input: Initial replication plan $\mathbf{P}$; Amount of available resources R; Topology T;
Output: Replication Plan $\mathbf{P}$
1 Initialize: $usage \leftarrow 0$;
2 Sort the set of tasks S_i of each operator O_i based on the OF increase, δ_{ij} , of tasks;
3 **if** $P = \emptyset$ **then**
4 **foreach** O_i **do**
5 Let p_{ik} be the node in S_i that has the largest OF increase δ_{ik};
6 $\mathbf{P} \leftarrow \mathbf{P} \cup \{p_{ik}\}$; $S_i \leftarrow S_i - \{p_{ik}\}$;
7 return $\mathbf{P}$
8 **if** $usage < R$ **then**
9 $Candidates \leftarrow \emptyset$;
10 **foreach** O_i **do**
11 Let p_{ik} be the node in S_i that has the largest OF increase δ_{ik} with a cost smaller than $R - usage$;
12 $Candidates \leftarrow Candidates \cup P \cup \{p_{ik}\}$;
13 $P_j \leftarrow$ **max_accuracy_plan**$(Candidates)$;
14 $S_j \leftarrow S_j - \{p_{jk}\}$; $\mathbf{P} \leftarrow P_j$; $usage \leftarrow usage + Cost(p_{jk})$;
15 Return $\mathbf{P}$;

Algorithm 3: STRUCTUREAWARE(R,T)

Input: The amount of available resources R; Topology T;
Output: Partial replication plan $\mathbf{P}$;
1 Initialize: decompose the complete topology T into sub-topologies: $TS_1, TS_2, \dots$;
2 $\mathbf{P} \leftarrow \emptyset$, $S_A \leftarrow \emptyset$, $usage \leftarrow 0$;
3 **foreach** *Sub-Topology* TS_i **do**
4 $P_i \leftarrow$ `PlanSubTopology` $(\emptyset, R, TS_i)$; $\mathbf{P} \leftarrow \mathbf{P} \cup P_i$;
5 $P_i' \leftarrow$ `PlanSubTopology` (P_i, R_i, TS_i);
6 $C_i \leftarrow Cost(P_i') - Cost(P_i)$; $\Delta_i \leftarrow \frac{OF_{P_i'} - OF_{P_i}}{C_i}$;
7 Put Δ_i into S_A in descending order;
8 $usage \leftarrow usage + Cost(P_i)$;
9 **while** $usage < R$ **do**
10 $LastUsage \leftarrow usage$; $j \leftarrow 1$;
11 **while** $j \leq |S_A|$ **do**
12 $\Delta_i \leftarrow j$th value in S_A; $j++$;
13 **if** $C_i + usage \leq R$ **then**
14 Use P_i' to replace P_i in $\mathbf{P}$;
15 Calculate new C_i, Δ_i. Insert Δ_i into S_A in descending order; break;
16 **if** $usage \leftarrow lastUsage$ **then** break;
17 Return $\mathbf{P}$;
Function: `PlanSubTopology` $(\mathbf{P}, R, T)$
18 **if** T *is a full topology* **then**
19 $\mathbf{P} \leftarrow$ PLANFULLTOPOLOGY$(\mathbf{P}, R, T)$;
20 **else** $\mathbf{P} \leftarrow$ PLANSTRUCTUREDTOPOLOGY$(\mathbf{P}, R, T)$;

larger OF increases, i.e. δ_{ik} , than other tasks in the topology and put them into P (lines $10-13$). The complexity of this algorithm is $O(N \cdot M)$, where the notations are defined in Section 4.1.

4.3 Solution for General Topology

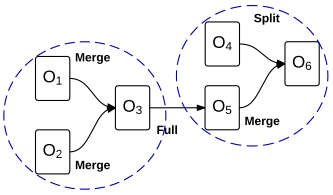


Fig. 4. Example of splitting a topology into sub topologies.

With the above algorithms, we divide a general topology into several sub-topologies and then use the corresponding algorithms according to the type of each sub-topology to generate the replication plans. We require that at least one partitioning function between any two neighboring sub-topologies is Full and the amount of sub-topologies is minimized. The reason behind this requirement is to make the selection of the replication segments in the sub-topologies independent from each other.

The split algorithm explores the topology using multiple depth-first searches (DFS). At the beginning, only the sink operator of the given topology is in the start point set SP. At each iteration, we will pick an operator, O_s, from SP and build a sub-topology by performing a DFS starting from O_s. If the DFS arrives at an operator O_i whose partitioning function is incompatible with the type of the current sub-topology, it will not further traverse O_i's downstream operators and O_i will not be added to the current sub-topology but instead be put into SP. Finally the algorithm will terminate if SP is empty. Figure 4 presents an example general topology, which is decomposed into two sub-topologies: $\{O_1, O_2, O_3\}$ and $\{O_4, O_5, O_6\}$.

We present details of the correlated-failure optimization algorithm for a general topology in Algorithm 3, which is referred to as the Structure Aware algorithm. The algorithm first decomposes the topology into sub-topologies which are either full topologies or structured topologies. Then the algorithm runs in multiple iterations. Within each iteration, it will try to get a replication plan from each sub-topology and select the one with the maximum profit density (lines $11-17$). The loop will be terminated when there is no more resource to replicate a complete MC-Tree. The algorithm's complexity is equal to $O(R \cdot N \cdot M^2 \cdot E)$, where the notations are defined in Section 4.1.

4.4 Optimization for Multi-Query Topology

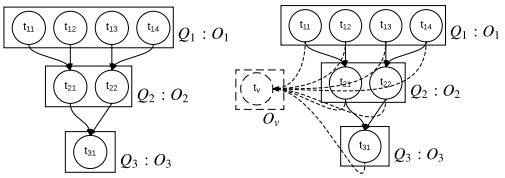
In this section, we present how to adapt the Structure Aware (SA) algorithm to handle multi-query topologies. Our first exploration is to change the optimization objective of the single-query SA from maximizing the OF of the single query to the weighted sum of the OF of every query in the topology, which is $\sum_{i=1}^{N} prt_i \cdot OF_i$ as described in Definition 2. Then we can directly use the single-query SA with the new optimization objective to generate PPA plan for multi-query topology. However, the deficiency of this approach is that, the search of failed MC-Trees in the single-query SA is conducted within the set of MC-Trees of the global topology. As the MC-Trees of different queries in the global topology could be of various sizes and structures,the confining of search space in the single-query SA would damage the efficiency of resource utilization when applying it on a multi-query topology.

For instance, there are in total 4 MC-Trees of the global topology depicted in Figure 5(a). If the priority of Q_1 and Q_2 are higher than that of Q_3, comparing to replicating an MC-Tree of the global topology, replicating an MC-Tree from Q_1 (e.g. $\langle t_{11} \rangle$) or Q_2 (e.g. $\langle t_{11}, t_{12}, t_{21} \rangle$) could bring higher OF increment with less resource consumption. Therefore, only consider MC-Trees of the global topology is suboptimal.

To solve this problem, we covert a multi-query topology T into a single-query topology T' by adding a virtual operator such that the MC-Trees of each query are also MC-Trees in the converted topology. In detail, we add a virtual independent-input operator O_v into T. The virtual operator O_v consists of only one virtual task and subscribes to the output streams of all the queries' sink operators in T. Figure 5(b) depicts the converted form of the topology in Figure 5(a). After the conversion, each MC-Tree (e.g. $\langle t_{11} \rangle$) of Q_1, Q_2 and Q_3 in the raw topology is transformed into an MC-Tree (e.g. $\langle t_{11}, t_v \rangle$) in the converted one.

The resource consumption of the virtual task t_v is set

as 0. Denoting the converted topology as as T' and the set of tasks in the sink operator of query Q_i as $SK_i = \{t_1, t_2, ..., t_{M_i}\}$. In T' the rate of each virtual stream between the task t_j in SK_i and the single virtual task in O_v is calculated as: $\lambda'_{t_j} = prt_i \cdot \frac{\lambda_{t_j}}{\sum_{t_k \in SK_i} \lambda_{t_k}}$, where λ_{t_k} represents the rate of the non-virtual output stream of task $t_k \in SK_i$. The advantage of the topology conversion is that, the output fidelity of the virtual operator O_v in T' is equal to the weighted sum of the output fidelities of queries, which is just our optimization objective, in the raw topology T,



(a) Before Conversion (b) After Conversion

Fig. 5. An example of adding virtual operator O_v to the raw multi-query topology which consists of query Q_1, Q_2 and Q_3. The directed edges depicted in dashed line denote the virtual sub-streams.

We perform sub-topology decomposition for each query (including the virtual sink operator) independently, which starts from the virtual sink operator and ends at the query's source operators. Note that sub-topologies from different queries may share operators. The operations after the decomposition are similar with the Structure Aware algorithm for single query topology. By incrementing the amount of available resource gradually, a local recovery plan is generated for each sub-topology, among which the one that brings the highest increment in the global OF per unit of resource is selected for active replication. Assuming that the number of queries is bounded by the number of operators in the topology, the time complexity of multi-query SA is $O(R \cdot N^2 \cdot M^2 \cdot E)$, where the notations are defined in Section 4.1.

5 Incremental Recovery for Passively-Replicated Tasks

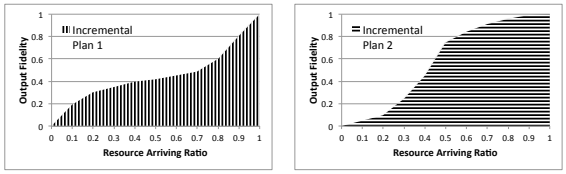
Correlated failures would usually incur the unavailability of a large amount of resources. Furthermore, it is unrealistic to assume an instant acquisition or recovery of sufficient resources to restore all the failed tasks. For instance, resolving correlated failure on a local cluster may involve solving the software or hardware problems, restarting the failed nodes, and adding them back to the DSPE. Even if the DSPE is running on a cloud environment and virtual resources can be easily allocated to replace the failed nodes, negotiating and acquiring a large amount of new resources would still incur a non-negligible latency for streaming data applications. In our experiences, while attaching new nodes to a deployed Storm cluster on Amazon EC2, the time interval between the arrivals of the first available node and the last one ranges from several minutes to tens of minutes, incrementing with the number of the newly attached nodes.

Even with a PPA plan, postponing the recovery of the passively replicated tasks until the arrival of all the necessary resources would incur a significant latency to a streaming data applications. We propose incremental recovery as an integral part of PPA to solve this problem. With incremental recovery, the recovery of the passively replicated tasks would be gradually scheduled based on the current availability of resources, such that the OF of the tentative outputs can be maximized as fast as possible. While the acquisition of new resources is ongoing, PPA generates an incremental recovery plan under the constraint of the amount of currently available resources. This plan enables the recovery of a subset of the failed MC-Trees. Once the arrivals of new resources are detected, another incremental recovery plan will be generated and executed. This process is repeated until all the passively replicated tasks involved in the failure are recovered.

5.1 Problem Definition

The process of incremental recovery spans for a period of time. Therefore, instead of acquiring the highest increment in OF for each incremental recovery plan when new resource arrives, we would like to maximize the average OF of the tentative outputs generated during the entire process of recovery, as the latter could represent the overall quality of tentative outputs. Before presenting our solution, we define the problem of optimizing incremental recovery for passively replicated tasks as follows:

Definition 3. *Denoting the instant when the failure is detected as t_s and the moment when the recovery is completed as t_e, the objective of incremental recovery is to maximize $\frac{\int_{ts}^{te} OF_t \cdot dt}{t_e - t_s}$, where OF_t is the output fidelity of the tentative outputs at timestamp t.*



(a) Incremental Plan : P_1 (b) Incremental Plan : P_2

Fig. 6. Examples illustrating the effects of incremental recovery plans.

A straightforward strategy is to gradually recover the failed tasks in a topological order, which prioritizes the recoveries of tasks whose upstream tasks have not failed or have already been recovered. However, this approach ignores the fact that MC-Trees can contribute to the final outputs if and only if all of their tasks are alive, thus it fails to maximize the average OF of the tentative outputs. The SA algorithm presented in Section 4 aims to maximize the OF of the PPA plan under a static resource constraint, which therefore cannot be readily applied to incremental recovery. Note the optimization of incremental recovery is not to maximize the OF at a given instant, but rather the average OF of the tentative outputs throughout the recovery. Figure 6 presents the OF values over recovery period under two possible incremental recovery plans. Although the OF of plan P_1 increases faster than P_2 in the early phase of the recovery, we prefer plan P_2 as it has a larger shadowed area. This example also infers that, while generating an incremental plan, maximizing the current OF should not be the only objective, and we should also take into account its effect on the subsequent recovery process. For instance, while generating local incremental plan, besides the OF increment and the resource consumption, the number of failed queries that share the recovered operators in each candidate plan also influences its recovery priority.

Algorithm 4: INCREMENTALRECOVERY(R, T, F)

Input: Recovery resources R; Topology T; Set of failed tasks: F;
Output: Incremental Plan $\mathbf{\Delta P}$;

1 Initialize: Decompose T into sub-topologies: $ST_1, ST_2, \dots$;
2 $\mathbf{\Delta P} \leftarrow \emptyset$; $S_1 \leftarrow \emptyset$; $S_2 \leftarrow \emptyset$; $usage \leftarrow 0$;
3 **foreach** *Sub-topology* ST_i **do**
4 $\quad \mathbf{\Delta P_i} \leftarrow$ IncrementSubTP(R, ST_i, F);
5 $\quad$ **if** *There is no complete MC-Tree in* ST_i **then**
6 $\quad\quad S_1 \leftarrow \mathbf{\Delta P_i}$;
7 $\quad$ **else**
8 $\quad\quad S_2 \leftarrow \mathbf{\Delta P_i}$;
9 **if** $S_1 \neq \emptyset$ **then**
10 $\quad$ Traverse S_1 in topological order and put the visited $\mathbf{\Delta P_i}$ into $\mathbf{\Delta P}$ while the resource is enough; Update S_1 and $usage$;
11 Sort local incremental plans in S_2 in the descending order of $OD(\mathbf{\Delta P_i})$;
12 **while** $usage < R$ **do**
13 $\quad LastUsage \leftarrow usage$; $j \leftarrow 1$;
14 $\quad$ **while** $j \leq |S_2|$ **do**
15 $\quad\quad \mathbf{\Delta P_i} \leftarrow j$th value in S_2; $j++$;
16 $\quad\quad$ **if** $Cost(\mathbf{\Delta P_i}) + usage \leq R$ **then**
17 $\quad\quad\quad \mathbf{\Delta P} \leftarrow \mathbf{\Delta P} \cup \mathbf{\Delta P_i}$;
18 $\quad\quad\quad \mathbf{\Delta P_i} \leftarrow$ IncrementSubTP(R, ST_i, F);
19 $\quad\quad\quad$ Insert $\mathbf{\Delta P_i}$ into S_2 in the descending order of $OD(\mathbf{\Delta P_i})$;
20 $\quad\quad\quad usage \leftarrow usage + Cost(\mathbf{\Delta P_i})$;
21 $\quad\quad\quad$ break;
22 $\quad$ **if** $usage = lastUsage$ **then** break;
23 Return $\mathbf{\Delta P}$;

Algorithm 5: INCREMENTSUBTP(R, ST, F)

Input: Recovery resources R; Topology T; Set of failed tasks: F;
Output: Incremental plan $\mathbf{\Delta P}$;

1 Initialize: $usage \leftarrow 0$;
2 **if** ST *is a Structured sub-topology* **then**
3 $\quad$ `IncrementStructuredTP` (R, ST, F)
4 **if** ST *is a Full sub-topology* **then**
5 $\quad$ `IncrementFullTP` (R, ST, F)

Function: `IncrementStructuredTP`(R, ST, F)

6 $S_u \leftarrow$ Set of the units split from ST;
7 **foreach** *Unit* $U_i \in S_u$ **do**
8 $\quad G_i \leftarrow$ Set of segments containing failed task;
9 $\quad$ **foreach** *segment* $g_i \in G_i$ **do**
10 $\quad\quad f_i \leftarrow$ the frequency that g_i is shared by MC-Trees in S_u;
11 **while** $usage \leq R$ **do**
12 $\quad Candidates \leftarrow \emptyset$;
13 $\quad$ **foreach** *Unit* $U_i \in S_u$ **do**
14 $\quad\quad$ **foreach** *failed segment* $g_i \in G_i$ **do**
15 $\quad\quad\quad$ Based on g_i, find a set of segments CG_i to build a complete MC-Tree in ST;
16 $\quad\quad\quad Candidates \leftarrow Candidates \cup CG_i$;
17 $\quad$ Find $CG_{opt} \in Candidates$ that maximizes: $\Delta OF' / \sum^{g_j \in CG_{opt}} \frac{Cost(g_j)}{f_j}$;
18 $\quad \mathbf{\Delta P} \leftarrow \mathbf{\Delta P} \cup CG_{opt}$;
19 $\quad usage \leftarrow usage + Cost\,(CG_{opt})$;
20 $\quad$ Update f_i for all the failed segments not included in CG_{opt};
21 Return $\mathbf{\Delta P}$;

Function: `IncrementFullTP`(R, ST, F)

22 $SO \leftarrow$ Set of operators in ST where all of their tasks are failed;
23 **while** $usage < R$ **do**
24 $\quad Candidates \leftarrow \emptyset$;
25 $\quad$ **if** $SO \neq \emptyset$ **then**
26 $\quad\quad$ **foreach** $O_i \in SO$ **do**
27 $\quad\quad\quad f_{ik} \leftarrow$ number of failed Mc-Trees that contain t_{ik}; $Candidates \leftarrow Candidates \cup \{t_{ik}\}$;
28 $\quad\quad \Delta P \leftarrow \Delta P \cup \{t_{ik}\}$, where t_{ik} has the amxmized $\frac{\Delta OF'_{ik} \cdot f_{ik}}{Cost(t_{ik})}$ in $Candidates$; Remove O_i from SO;
29 $\quad\quad usage = usage + Cost\,(t_{ik})$ Continue;
30 $\quad$ **foreach** O_i **do**
31 $\quad\quad t_{ik} \leftarrow$ the failed task of O_i whose recovery maximizes $\frac{\Delta OF'_{ik} \cdot f_{ik}}{Cost(t_{ik})}$;
32 $\quad\quad Candidates \leftarrow Candidates \cup \{t_{ik}\}$;
33 $\quad \Delta P \leftarrow \Delta P \cup \{t_{ik}\}$, where t_{ik} has the largest profit density in $Candidates$;
34 Return $\mathbf{\Delta P}$;

5.2 Optimizing Incremental Recovery

The algorithm that generates incremental recovery plan for global topology is presented in Algorithm 4. This algorithm starts by decomposing the global topology into a set of sub-topologies (line 1). For each sub-topology ST_i, the algorithm IncrementSubTP (Algorithm 5) is called to return a local incremental plan $\mathbf{\Delta P_i}$ that recovers a subset of failed MC-Trees in ST_i (line 4). The topology is disconnected if there exists sub-topology that has no MC-Trees executing normally. To ensure that the global topology can be connected, we store the local incremental plans returned for these disconnected sub-topologies in set S_1 and traverse S_1 in topological order to add them into the global incremental plan (lines $5-6$, $9-10$). Then the partial incremental plans generated for the other sub-topologies are sorted with the metric $OD(\mathbf{\Delta P_i})$, which is defined as $\frac{\Delta OF_i}{Cost(\mathbf{\Delta P_i})}$, where ΔOF_i denotes the OF increment by adding the local incremental plan $\mathbf{\Delta P_i}$ into the global plan $\mathbf{\Delta P}$ and $Cost(\mathbf{\Delta P_i})$ returns the resource consumption of $\mathbf{\Delta P_i}$. In the next while loop (line 12), we gradually increase the amount of available resource. The global incremental plan is expanded by adding the local incremental plan $DeltaP_i$ which has the largest $OD(\mathbf{\Delta P_i})$ with affordable amount of resource consumption. (lines $14-17$). The local incremental plan for sub-topology ST_i will be updated and added back to the plan pool S_2 (line $18-19$).

IncrementSubTP (Algorithm 5) consists of two functions: *IncrementStructuredTP(R, ST, F)* for structured topologies and *IncrementFullTP(R, ST, F)* for full topologies. The idea of *IncrementStructuredTP(R, ST, F)* is to expand the failed segments in a greedy approach to generate a set of candidate plans that consists of failed MC-Trees within the sub-topology. Details of the expansion are explained in Section 4.1. Considering that failed segments can be shared by multiple failed MC-Trees whose recoveries consume different amount of resources and bring different OF increments, we use a new metric to denote the recovery priorities of the candidate plans. This metric is calculated as $\Delta OF'_i / \sum^{g_i \in CG_j} \frac{Cost(g_i)}{f_i}$, where $\Delta OF'_i$ represents the local OF increment in the sub-topology ST by recovering CG_j, g_i is a failed segment in CG_j with its cost denoted as $Cost\,(g_i)$. The frequency that g_i is shared by the failed MC-Trees in ST is denoted as f_i. The candidate plan with the highest recovery priority is returned.

As for a *Full* sub-topology ST, by denoting the number of the failed MC-Trees that share a failed task $t_{jk} \in O_j$ as f_{ik}, f_{ik} can be calculated as $\left(\prod^{O_i \in ST} p_i - \prod^{O_i \in ST} e_i\right) / e_j$, where p_i is the degree of parallelization of O_i and e_i denotes the number of the failed tasks in O_i. Similar to *IncrementStructuredTP(R, ST, F)*, we use the metric $\frac{\Delta OF'_{ik} \cdot f_{ik}}{Cost(t_{ik})}$ to generate the recovery order of the failed tasks in a *Full* sub-topology. The reason that the sharing frequency f_{ik} is incorporated in the above two metrics is that we prefer the recovery of a MC-Tree in the current incremental plan to benefit more failed MC-Trees in the subsequent recoveries.

6 SYSTEM IMPLEMENTATION

6.1 Framework

We implemented our system on top of Storm. In comparing to Spark Streaming, which processes data in a micro-batching approach, Storm will process an input tuple once it arrives and thus can achieve sub-second end-to-end processing latency. As shown in Figure 7, the nimbus in the Storm

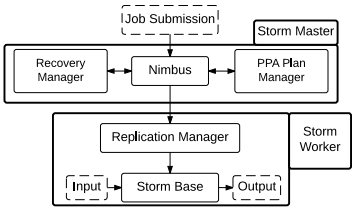


Fig. 7. System Framework

master node assigns tasks to the Storm worker nodes and monitoring the failures. On receiving a job, the nimbus will transfer the query topology to the PPA plan manager, which will generate a PPA recovery plan under the constraint of resource usage of active replication. The PPA recovery plan consists of two parts: a completely passive standby plan and a partially active replication plan. Based on the PPA recovery plan, the replication manager in the worker nodes will create checkpoints to passively replicate the whole query topology. Checkpoints will be stored onto a set of standby nodes. The replication manager will create active replicas for the tasks that are included in the partially active replication plan. The active replicas can support fast failure recovery and will also be deployed onto the standby nodes.

Once a failure is detected by the nimbus, The recovery manager in the Storm master node will decide how to recover the failed tasks based on the PPA replication plan. For the tasks that are actively replicated, the recovery manager will notify the nimbus to recover them using their active replicas such that the tentative results could be produced as soon as possible. The failed tasks that are passively replicated will be recovered with their latest checkpoints.

6.2 PPA Fault Tolerance

Passive Replication. In PPA, checkpoints of the tasks are periodically created and stored at the standby nodes. We adopted the batch processing approach [28] to guarantee the processing ordering during recovery is identical to that before the failure. With this approach, input tuples are divided into a consecutive set of batches. A task will start processing a batch after it receives all its input tuples belonging the current batch. This is ensured by waiting a batch-over punctuation from each of its upstream neighboring tasks. Tuples within a batch are processed in a predefined round-robin order. The effect of batch size on the system performance has been researched in previous work [6].

A single point failure can be recovered by restarting the failed task, loading its latest checkpoint and replaying its upstream tasks' buffered data. The downstream tasks will skip the duplicated output from the recovering task until the end of the recovery phase. While recovering a correlated failure, if a task and its upstream neighboring task are failed simultaneously and its checkpoint is made later than its upstream peers', the recovery of the downstream task can only be started after its upstream peer has caught up with the processing progress. In other words, synchronizations have to be carried out among the neighboring tasks.

Active Replication. If task t has an active replica t', the output buffer of t' will store the output tuples produced by processing the same input in the same sequence as t does. The downstream tasks of t will subscribe the outputs from both t and t'. By default, the output of t' is turned off. To reduce the buffer size on t', its primary, t, will periodically notify t' about the latest output progress and the latter can then trim its output buffer. If t is failed, t' will start sending data to the downstream tasks of t. The downstream tasks will eliminate the duplicated tuples from t' by recognizing their sequence numbers. The batch processing strategy can guarantee an identical processing order between the primary and active replica of a task.

Tentative Outputs. As checkpoint-based recovery requires replaying the buffered data and synchronizations among the connected tasks and hence incurs significant recovery latency, PPA has the option to continue producing tentative results once the actively replicated tasks are recovered. Recall that during normal processing, a task will only start processing a batch after receiving the batch-over punctuations from all of its upstream neighboring tasks. If any of its upstream neighboring tasks fails, the recovery manager in the Storm master node will generate the necessary batch-over punctuations for those failed tasks, such that a batch could be processed without the inputs from the failed tasks and tentative outputs will be generated with an incomplete batch. After the failed tasks are recovered, the recovery manager will stop sending the batch-over messages for them such that the downstream tasks will wait for the batch contents from the recovered tasks before processing a batch. After all the failed tasks are recovered, the topology will start generating accurate outputs. In this paper, we assume the adoption of similar techniques proposed in [3] to reconcile the computation state and correct the tentative outputs and leave the implementation of these techniques as our future work.

Statistics Collecting. To conduct incremental recovery, we added a statistic collector in recovery manager which periodically collects workload statistics of each task and nodes. A timer thread running in the statistics collector periodically sends the requests for workload statistics to the processing tasks, which will then send its own CPU usage during the last statistics period to the statistics collector. The CPU usages of the tasks and nodes are denoted as a percentage between 0 and 100%. The collected statistics are stored in ZooKeeper.

7 Evaluation

The experiments are run over the Amazon EC2 platform. We build a cluster consisting of 36 instances, of which 35 m1.medium instances are used as the processing nodes and one c1.xlarge instance is set as the Storm master node. Heartbeats are used to detect node failures in a 5-second interval. The recovery latency is calculated as the time interval between the moment that the failure is detected and the instant when the failed task is recovered to its processing progress before failure. The processing progress of a task is defined as a vector. Each field of the progress vector contains the sequence number of the latest processed tuple from a specific input stream of the task. A failed task is marked as recovered if the values of all the fields in its current progress vector are larger than or equal to the values of the corresponding fields of the progress vector before failure. Additional information of the experiment configuration will be presented in the following sections.

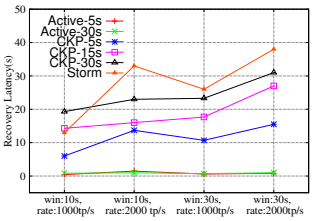


Fig. 8. Single failure.

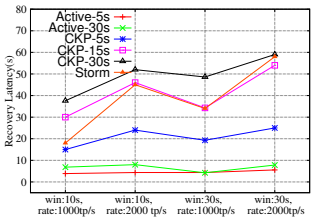


Fig. 9. Correlated failure.

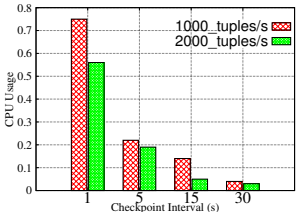


Fig. 10. Checkpointing cost.

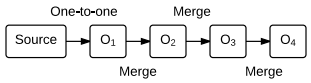


Fig. 11. Topology used in the experiments of recovery efficiency.

7.1 Recovery Efficiency

Firstly, we study the recovery efficiencies of different fault-tolerance techniques, including checkpoint, which is used in Spark Streaming, source replay, which is the default fault-tolerance technique in Storm, and active replication.

We implement a topology that consists of 1 source operator and 4 synthetic operators. The structure of this topology is depicted in Figure 11. The source operator consists of totally 16 tasks deployed on 4 nodes. All of the source tasks produce input tuples in a specified rate (1000 tuples/s or 2000 tuples/s). The degree of parallelization of operators O_1, O_2, O_3 and O_4 are set as 8, 4, 2 and 1 respectively. Each task in O_1 receives inputs from two source tasks and each task in O_2, O_3 and O_4 receives inputs from two upstream neighboring tasks. The 15 synthetic tasks are evenly distributed among 15 nodes. In addition, there are another 15 nodes used as the backup nodes to store the checkpoints and to run the active replicas. Each of the four synthetic operators maintains a sliding window whose sliding step is set as 1 second and the window interval varies from 10 seconds to 30 seconds. The state of each task of a synthetic operator is composed by the input data within the current window interval. The largest state size of a task is equal to the multiplication of the input rate and the window interval. The selectivity of each synthetic operator is set as 0.5.

In Figure 8 and Figure 9, Active-Xs denotes the case that output trimming for an active replica is conducted every X seconds. CKP-Xs represents the case that the checkpoint interval is set as X seconds.

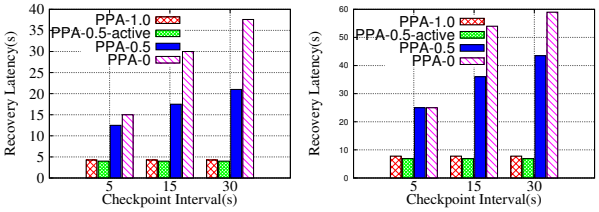
Single Node Failure. Figure 8 presents the recovery latencies of single node failures. For active replication, we vary the frequency of synchronizing the replica with its primary task. One can see that the active approach has much lower recovery latency than the passive approaches and the changes of window intervals and input rates have little influence. On the other hand, the recovery latencies with both Checkpoint and Storm increase proportionally with the input rate, as a higher input rate results in more tuples to be replayed during recovery for both approaches. Furthermore, the recovery latency with Checkpoint increases with the checkpoint interval. This is because the number of tuples that need be reprocessed to recover the task state increases with the checkpoint interval.

As Storm will have to replay more source data with longer window intervals, one can see that the recovery latency of Storm with 30-second window is higher than those with 10-second window. Another factor that influences the recovery latency of Storm is the location of the failed task in the topology, because the replayed tuples will be processed by all the tasks located between the tasks of the source operator and the failed tasks. Thus the recovery latency of Storm is higher than that of Checkpoint in most of the cases in this experiment. Here, we record the recovery latencies of tasks in different locations within the topology in Storm and report their average values.

Correlated Failure. We inject a correlated failure by killing all the nodes on which the primary replicas of the tasks are deployed. In Figure 9, one can see that the recovery latency of active replication is much lower than the passive approaches and stays stable with various the window sizes and input rates. Furthermore, active replication with a shorter synchronization period leads to faster failure recovery. On the other hand, the recovery latency of Checkpoint and Storm grows rapidly with the increment in window size and the input rate.

By comparing the results presented in Figure 8 and Figure 9, it can be seen that the recovery latency with active replication is lower than the passive approaches and is relatively stable under the scenarios of various input rates and window intervals. Moreover, the benefits of using active replication are larger in the case of correlated failure than that in the case of single node failure.

The latency of failure recovery with checkpoint can be reduced by setting a short checkpoint interval. However, the resource usage of maintaining checkpoints varies with different checkpoint intervals. Figure 10 presents the ratio of the CPU usage of maintaining checkpoint to that of normal computation within a task. We can see that the CPU usage of maintaining checkpoints increases quickly with shorter checkpoint intervals and making checkpoint with very short intervals such as one second is prohibitively expensive. Although active replication consumes more recourses than the passive approach, the low-latency recovery of active replication makes it meaningful in the context of MPSPEs.



(a) rate:1000 tuples / sec (b) rate:2000 tuples / sec

Fig. 12. Recovery latency of a correlated failure with PPA, window length : 30 seconds. PPA-0.5-active indicates the recovery latency of actively replicated tasks in plan PPA-0.5.

Recovery with PPA. We conducted experiments to study the performance of PPA with three active replication plans denoted as PPA-1.0, PPA-0.5 and PPA-0 respectively. These PPA plans consume various amount of resources for active replication. In PPA-1.0, all the tasks in the topology are actively replicated. PPA-0.5 is a hybrid replication plan where only half of the tasks have active replica. PPA-0 is

a purely passive replication plan where all the tasks are only replicated with checkpoint. The results are presented in Figure 12. As the failed tasks with active replicas are recovered faster than those using checkpoints, the overall recovery latency of PPA-0.5 is higher than that of PPA-1.0 but lower than that of PPA-0. Note that with PPA-0.5, the recovery latencies of tasks with active replicas (denoted as PPA-0.5-active in Figure 12) are much lower than that of recovering all the failed tasks (denoted as PPA-0.5 in Figure 12). The recoveries of PPA-0.5-active consume slightly less time than PPA-1.0, this is because the number of actively replicated tasks recovered in PPA-0.5-active is only the half of that in PPA-1.0. This set of experiments illustrate that the purely active replication plan outperforms the hybrid and purely passive plan regarding the recovery latency. With a hybrid plan, as the recoveries of actively replicated tasks finish earlier than that of the passively replicated ones, PPA can generate tentative outputs without waiting for the slow recoveries of passively replicated tasks.

7.2 Tentative Output Quality

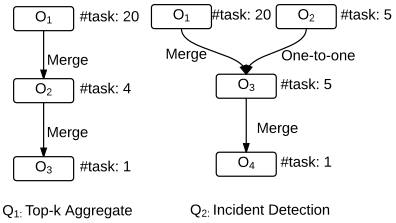
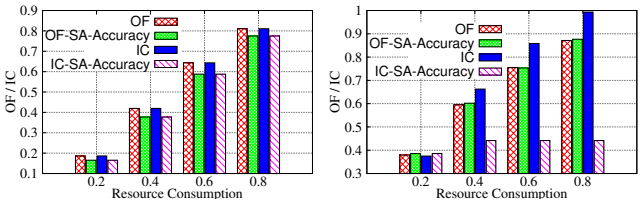


Fig. 13. *Top-k* aggregate query(Q_1) and incident detection query(Q_2).

In both this section and Section 7.3, the resource consumption of a PPA plan denotes the ratio of the amount of resources for active replication to the total resource consumption of the topology without active replication.

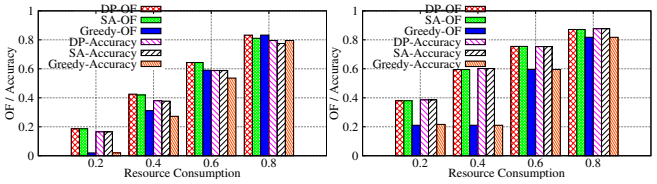
Q_1 is a sliding-window query that calculates the *top-100* hottest entries of the official website of World Cup 1998. The input dataset consists of in total $73,291,868$ access records. In the experiments, we replay the raw input stream in a rate which is 48 times faster than the original data rate. We implement this query as a topology that conducts hierarchical aggregates, which is a common computation in data stream applications. The structure of this topology is depicted in Figure 13. Tasks in O_1 split the input stream into a set of consecutive slices, each consisting of 100 tuples, and calculate their aggregate results. For every 100 input tuples, tasks in O_2 will conduct a merge computation and send the results to the single task in O_3, which periodically updates the globally *top-100* entries.

Q_2 is a sliding-window query detecting incidents that incur traffic jams. The window interval is 5 minutes and the sliding step is 10 seconds. As relevant datasets for this query are not publicly available due to privacy considerations, we generate a synthetic dataset in a community-based navigation application. There are two streams in this dataset: the user-location stream and the incident stream. The rate of the user-location stream is set as 20,000 location records per second. The incident stream is composed of user-reported incident events and the time interval between two consecutive incidents is set as 2 seconds. We distribute 100,000 users among 1000 virtual road segments following the Zipfian distribution (with parameter $s = 0.5$). The probability of incident within a segment is set to be proportional to the number of users located on it. The topology of Q_2 is presented in Figure 13. Tasks in O_1 receive the user-location records and calculate the average speed of each segment per second. Tasks in O_2 combine the user-reported incident events into distinct incident events. O_3 joins the segment-speed stream from O_1 and the distinct-incident stream from O_2. The outputs of tasks in O_3 are the incidents that incur traffic jams. O_4 aggregates the outputs of O_3.



(a) Query: Q1. (b) Query: Q2.

Fig. 14. Comparing the values of OF/IC and the query accuracy. OF-SA-Accuracy (or IC-SA-Accuracy) denotes the actual query accuracies of the PPA plans generated using the structure-aware(SA) algorithm with OF (or IC) as the optimization metric.

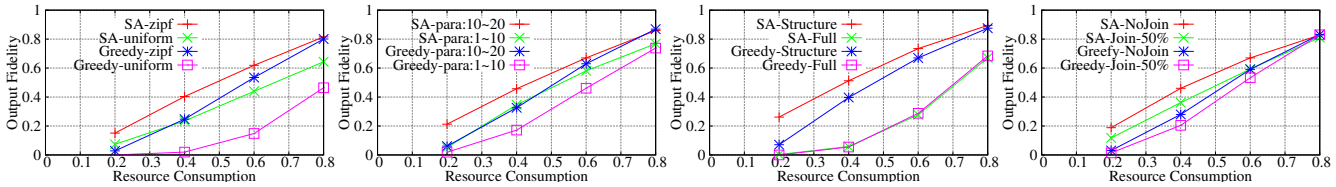


(a) Query: Q1. (b) Query: Q2.

Fig. 15. Comparing the values of OF and the actual query accuracies of the PPA plans which are generated by DP, SA and Greedy.

Validation of the OF metric. In this set of experiments, we compare OF with the Internal Completeness (IC) metric proposed in [4], which measures the fraction of the tuples that are expected to be processed by all the tasks in case of failures compared to the case without failures. A fundamental difference between OF and IC is that, OF takes the correlations of task's input streams into account.

By denoting the tentative outputs as S_T and the accurate outputs of Q_1 as S_A, we define the query accuracy of Q_1 as: $\frac{|S_T \bigcap S_A|}{|S_A|}$. Figure 14(a) shows the OF (or IC) values and the actual query accuracies of the PPA plans generated using the OF (or IC) metric. The results show that both OF and IC provide good predictions of the accuracy of typical top-k queries. This is because both OF and IC provide accurate estimations of the completeness of the inputs for aggregate queries, such as top-k, and such queries' output accuracies highly depend on the completeness of their inputs. The accuracy function of Q_2 is defined as $\frac{|I_T \bigcap I_A|}{|I_A|}$, where I_T is the set of tentative incidents generated with correlated failure and I_A is the set of accurate incidents generated without failure. As shown in Figure 14(b), the accuracy values are generally quite close to the values of OF. On the other hand, with more available resources, we can generate PPA plans with higher IC values. However, such plans do not have higher query accuracies. This is because IC fails to consider the correlation of tasks' input streams and hence cannot provide a good accuracy prediction for queries with joins. This result clearly indicates the importance of distinguishing join operators in predicting output accuracies.



(a) Workload Skewness (b) Parallelization (c) Full Partitioning (d) Join

Fig. 16. Comparing OF of SA and Greedy algorithm with random single-query topologies of various specifications, the number of operators is a random integer between 5 and 10. (a): The workloads of tasks within an operator are distributed in uniform or Zipfian distribution (with parameter $s = 0.1$). (b): The degree of operator parallelization is a random number between different ranges. (c): Topologies are either structured topology or full topology. (d): The fraction of join operators in the topologies is set as 0 or 50%.

Comparing Various Algorithms. To study the performance of the structure aware algorithm(SA), we also propose a Greedy algorithm that generates PPA plan based on the heuristic that tasks whose failures incur higher OF damages to the tentative outputs than the others should be assigned of higher priorities for recovery. More details of this algorithm are presented in [25]. In this set of experiments, we generate PPA plans for Q_1 and Q_2 using the dynamic programing algorithm(DP), the structure-aware algorithm(SA) and the Greedy algorithm respectively and compare their performances. Results presented in Figure 15 show that SA is quite close to DP, which generates the optimal PPA plan, in both OF and the actual query accuracy. Greedy has the worst performance in the results of both queries. This is because Greedy fails to consider that only complete MC-Trees can contribute to the query outputs.

7.3 Random synthetic topology

To conduct a comprehensive performance study of PPA algorithms with various types of topologies, we implemented a random topology generator which can generate topologies with different specifications including the number of operators, the degree of operator parallelization, the distribution of operator partitioning methods, the fraction of join operators in the topology and workload distribution of tasks within an operator. The generation of topology starts from the creation of a sink operator, from which the topology grows according to the user-specified parameters. For each set of topology specifications, we generate 100 synthetic topologies and use them as the inputs of SA algorithm and the Greedy algorithm to compare their performances in terms of OF. Due to the prohibitive complexity of the dynamic programing algorithm, we cannot complete it for this set of experiments within a reasonable time so we do not include it here. Query accuracies are not compared in this set of experiments, as we cannot derive the actual output accuracies for these randomized synthetic topologies.

Single-Query Topology. In Figure 16, all the topology consists of one query. We vary the topology specifications, such as workload skewness, parallelization degrees of operators, topology structure and the ratio of join operators, to compare the performances of SA and the Greedy algorithm. One can see that, SA outperforms the Greedy algorithm in all the combinations of topology specifications and active replication ratios. This is because the Greedy algorithm does not consider whether the actively replicated tasks in the generated PPA plan can compose complete MC-Trees.

Multi-Query Topology. Figure 17 presents the experimental results with synthetic multi-query topologies. As Greedy is agnostic to the topology structure, it falls behind the multi-query SA in all the experimental settings. Among

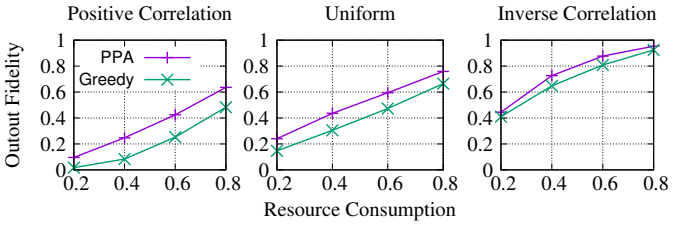
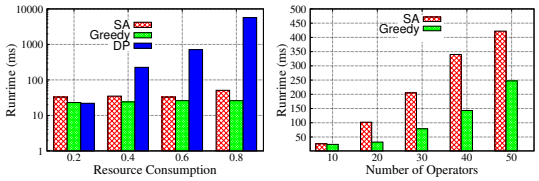


Fig. 17. Comparing OF of SA and Greedy. The number of operators is a random integer between 10 and 20, and the number of queries is a random integer between 2 and 10. The workloads of tasks within an operator follows a uniform distribution. From left to right, the figures depicts the cases that the priorities of queries are set as positively correlated to their resource consumptions, uniformly distributed or inversely correlated to their resource consumptions.

the three cases, multi-query SA has the greatest performance advantage over Greedy when the query priority is set as positively correlated with its resource consumption. As in this case, recovering MC-Trees with larger size results in a higher increment in OF, while Greedy falls short in constructing such MC-Trees in its plan. Compared to the case of positive correlation, the performance advantage of SA over Greedy decrements in the case of uniform distribution and shrinks even further in the case of inverse correlation. This is because Greedy tends to fully recover the smaller MC-Trees, which contribute more to the OF of tentative outputs in the last two cases.

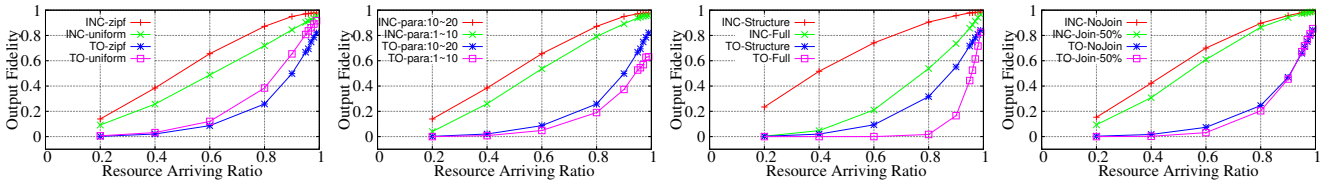


(a) Query: Q1 (b) Synthetic Topology

Fig. 18. Comparing the runtime of the PPA algorithms.

7.4 Runtime of Plan Generation

This set of experiments study the runtime of the three algorithms proposed for PPA. All the experiments are repeated for 100 times and the average runtime is plotted in Figure 18. Figure 18(a) presents the runtime results with Query Q_1 as the input topology, which shows that the runtime of the dynamic programming(DP) algorithm is up to two order of magnitude higher than that of SA and Greedy even with a small topology that consists of 20 MC-Trees. We also compared the runtime of SA and Greedy with larger synthetic topologies, where the number of operators ranges from 10 to 50. The degree of parallelization of operator is set as a random integer between 10 and 20. The resource consumption is set as 0.8. As one can see in Figure 18(b), both SA and Greedy can finish plan generation within half a second. Greedy has a shorter runtime as its search space



(a) Workload Skewness (b) Parallelization (c) Full Partitioning (d) Join

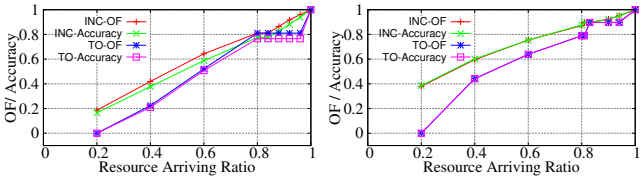
Fig. 19. Comparing OF of the INC and TO algorithms with random single-query topologies of various specifications, the number of operators is a random integer between 5 and 10. (a): The workloads of tasks within an operator are distributed in uniform or Zipfian distribution (with parameter $s = 0.1$). (b): The degree of operator parallelization is a random number between different ranges. (c): Topologies are either structured topology or full topology. (d): The fraction of join operators in the topologies is set as 0 or 50%.

is much smaller then PPA. From this set of experiments, we can conclude that SA is fast enough as the foundation for conducting dynamic plan adaptation.

7.5 Incremental Recovery

To study the performance of the incremental recovery algorithm (INC), we compare it with a baseline algorithm, which is referred to as TO. TO schedules the recoveries of the failed tasks following a topological order, it starts by adding the failed tasks of the source operators into a task pool. While gradually increasing the amount of available resources, it selects the task in the task pool whose independent failure causes the greatest damage to the OF of the tentative outputs. A non-source task is added into the task pool if all of its failed upstream neighbors are recovered. In this set of experiments, we use the random synthetic queries and the two real queries described in Section 7.2.

Figure 19 presents the experimental results with the synthetic topologies. In each run, we generate 100 synthetic topologies. The results are collected in the situation that all the tasks are failed in the correlated failure and the arriving of new resources follows a uniform distribution. One can see that, INC largely outperforms TO in all the experimental settings, especially in the phase of the recovery. This is because TO schedules the recovery following the topological order, which results in that only very few MC-Trees are recovered while the amount of available resources is small. Results presented in Figure 19(c) shows that the incremental plans that INC generates for structured topology have higher OF then those that INC generates for full topology. This is because the failure of any task in a full topology incurs output lost for all of its upstream neighboring tasks and damages the input quality of all the tasks in its downstream neighboring operators.



(a) Query: Q1. (b) Query: Q2.

Fig. 20. Comparing the OF and the actual query accuracies of the incremental recovery plans generated by INC and TO algorithms, respectively. Query Q_1 and Q_2 are presented in Figure 13.

Figure 20 presents results of the two real queries using the crawled tweets as inputs, which show that INC generates incremental plans with better OF performance than that of TO. Furthermore, one can see that the actual accuracy of tentative results generated with the incremental plans are very similar to their OF for both INC and TO, which exhibits the validity of output fidelity as a suitable accuracy metric during incremental recovery.

8 Related Work

Fault-tolerance in SPE. Traditional fault-tolerance techniques for SPEs could be categorized as passive [13], [27], [17] and active approaches [13], [3], [12]. The technique of delta checkpoint [14] is used to reduce the size of checkpoints. The authors in [9] proposed techniques to reduce the checkpoint overhead by minimizing the sizes of queues between operators, which are part of the checkpoints. [19] proposed to utilize the idle period of the processing nodes for active replication. Such optimizations are compatible to our PPA scheme and can be employed in our system.

Spark Streaming [28] uses Resilient Distributed Dataset (RDD) to store the states of processing tasks. In case of failure, RDDs can be restored from checkpoints or rebuilt based on its lineage. In other words, it adopts both the checkpoint-based and replay-based approaches. The works in [7], [22] explore optimistic recovery for fixpoint algorithms used in data mining. After failure, optimistic recovery utilizes a user-defined compensate function to create a consistent state to resume the execution. To reduce recovery latency, authors in [5], [28] proposed to use parallel recovery. With parallel recovery, multiple tasks can be launched to recover a failed task and each of them is recovering a partition of the failed one to shorten the process of passive recovery. However, with a correlated failure, a large number of failed tasks need to be recovered simultaneously. Then the degrees of parallel recovery would be constrained.

Hybrid fault-tolerance approaches are studied in [27], [11]. In [27], the objective is to minimize the total cost by choosing a passive fault-tolerance strategy, including upstream buffering, local checkpoint and remote checkpoint. [11] uses either active replication or checkpoint as the fault-tolerance approach for an operator. The work in [29] considers task overloading, referred to as "transient" failure, caused by temporary workload spikes. Upon a transient failure of a task, its active replica will be used to generate low-latency output. Different from these approaches, the trade-off of our work is between resource consumption and result accuracy facing correlated failures.

Tentative Outputs. Borealis [3] uses active replication for fault tolerance and allows users to trade result latency for accuracy. Compared with Borealis, PPA explores more on optimizing the accuracy of tentative results. Previous work [4] attempts to dynamically assign computation resources between primary computation and active replicas to achieve trade-offs between system throughput and fault-tolerance guarantee. Their accuracy model, IC, does not consider the correlation of processing tasks' inputs streams,

which is shown to be inadequate in our experiments.

A fault injection-based approach is presented in [15] to evaluate the importance of the computation units to the output accuracy, which only considers independent failures. Zen [21] optimizes operator placement within clusters under a correlated failure model. As operator placement is orthogonal to the planning of active replications, their techniques can also be employed as a supplement to PPA.

Incremental Recovery DSPEs, such as Flink [1] and Storm [26], adopt a blocking recovery approach in the sense that the recovery would be blocked until sufficient new resources are acquired. However, with such an approach, the resources which arrive earlier than the other would be idled before the recovery is started. There exist DSPEs [5], [18], [24] that support expanding the scope of recovery following the pace of arriving resources. However, the optimization of incremental recovery plan is not considered in [5], [18]. The work in [24] optimizes the recovery schedule of the failed queries. Different from PPA that optimizes the quality of tentative outputs, [24] only considers completely recovering each individual query, hence it is unsuitable for stream applications that prefer tentative outputs.

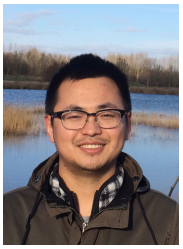
Failure in Clusters. Previous studies found that failure rates vary among different clusters and the number of failures is in general proportional to the size of the cluster [23]. Correlated failures do exist and their scopes could be quite large [10], [20]. Hence considering correlated failure is inevitable for a MPSPE that supports low-latency and nonstop computations.

9 Conclusion

We present a passive and partially active (PPA) fault-tolerance scheme for MPSPEs. In PPA, a partially active replication plan is optimized to maximize the accuracy of tentative outputs after failure. We also propose incremental recovery to schedule the recoveries of passively replicated tasks to optimize the average output accuracy during failure recovery. The experimental results indicate that, upon a correlated failure, PPA can start producing tentative outputs up to 10 times faster than the completion of recovering all the failed tasks. Hence PPA is suitable for applications that prefer tentative outputs with minimum delay. The experiments also show that our structure-aware algorithms can achieve up to one order of magnitude improvements on the qualities of tentative outputs in comparing the greedy algorithm that is agnostic to query topology structures. Therefore, to optimize PPA, it is critical to take advantage of the knowledge of the query topology's structure.

References

[1] http://flink.apache.org/.
[2] B. Babcock, M. Datar, et al. Load shedding for aggregation queries over data streams. ICDE'04.
[3] M. Balazinska, H. Balakrishnan, et al. Fault-tolerance in the borealis distributed stream processing system. *ACM Trans. Database Syst*, 2008.
[4] P. Bellavista, A. Corradi, et al. Adaptive fault-tolerance for dynamic resource provisioning in distributed stream processing systems. In *EDBT'14*.
[5] F. Castro, M. Raul, et al. Integrating scale out and fault tolerance in stream processing using operator state management. SIGMOD'13.
[6] T. Das, Y. Zhong, et al. Adaptive stream processing using dynamic batch sizing. SOCC '14.
[7] S. Dudoladov, C. Xu, et al. Optimistic recovery for iterative dataflows in action. SIGMOD '15.
[8] O. Goldschmidt, D. Nehme, and G. Yu. Note: On the set-union knapsack problem. *Naval Research Logistics (NRL)*, 41(6):833–842, 1994.
[9] Y. Gu, Z. Zhang, et al. An empirical study of high availability in stream processing systems. Middleware'09.
[10] T. Heath, R. P. Martin, et al. Improving cluster availability using workstation validation. SIGMETRICS'02.
[11] T. Heinze, M. Zia, et al. An adaptive replication scheme for elastic data stream processing systems. DEBS '15.
[12] J.-H. Hwang, U. Cetintemel, and others. Fast and highly-available stream processing over wide area networks. ICDE'08.
[13] J.-H. Hwang et al. High-availability algorithms for distributed stream processing. ICDE'05.
[14] J.-H. Hwang, Y. Xing, et al. A cooperative, self-configuring high-availability solution for stream processing. ICDE'07.
[15] G. Jacques-Silva, B. Gedik, et al. Fault injection-based assessment of partial fault tolerance in stream processing applications. DEBS'11.
[16] J. Kang, J. F. Naughton, et al. Evaluating window joins over unbounded streams. ICDE'03.
[17] Y. Kwon, M. Balazinska, et al. Fault-tolerant stream processing using a distributed, replicated file system. VLDB '08.
[18] A. Martin, A. Brito, and C. Fetzer. Scalable and elastic realtime click stream analysis using streammine3g. DEBS '2014.
[19] A. Martin, C. Fetzer, and A. Brito. Active replication at (almost) no cost. SRDS'11.
[20] S. Nath, H. Yu, Gibbons, et al. Subtleties in tolerating correlated failures in wide-area storage systems. NSDI'06.
[21] B. Nikhil, B. Ranjita, et al. Towards optimal resource allocation in partial-fault tolerant applications. In *INFOCOM'08*.
[22] S. Schelter, S. Ewen, et al. "all roads lead to rome": Optimistic recovery for distributed iterative data processing. CIKM '13.
[23] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance computing systems. DSN'06.
[24] L. Su and Y. Zhou. Progressive recovery of correlated failures in distributed stream processing engines. EDBT '17.
[25] L. Su and Y. Zhou. Tolerating correlated failures in massively parallel stream processing engines. In *ICDE'16*.
[26] A. Toshniwal, S. Taneja, et al. Storm@twitter. SIGMOD '14.
[27] P. Upadhyaya, Y. Kwon, et al. A latency and fault-tolerance optimizer for online parallel query plans. SIGMOD'11.
[28] M. Zaharia et al. Discretized streams: Fault-tolerant streaming computation at scale. SOSP '13.
[29] Z. Zhang, Y. Gu, et al. A hybrid approach to high availability in stream processing systems. ICDCS '10.
[30] J. Zhou et al. Advanced partitioning techniques for massively distributed computation. SIGMOD'12.



Li Su is a Postdoctoral Researcher in the Department of Mathematics and Computer Science, University of Southern Denmark. He obtained his PhD in Computer Science at the University of Southern Denmark, in 2016. His research interests include distributed stream processing and large-scale data management.

Yongluan Zhou is an Associate Professor in the Department of Computer Science at the University of Copenhagen. Before that, he had worked at the University of Southern Denmark and cole Polytechnique Fdrale de Lausanne. He obtained his PhD in Computer Science at the National University of Singapore. His research interests include stream processing, query processing, query optimization, and distributed systems.